# A TRIDENT SCHOLAR PROJECT REPORT

NO. 446

Fast, Distributed Algorithms in Deep Networks

by

Midshipman 1/C Ryan J. Burmeister, USN

UNITED STATES NAVAL ACADEMY
ANNAPOLIS, MARYLAND

USNA-1531-2

U.S.N.A. — Trident Scholar project report; no. 446 (2016)

# FAST, DISTRIBUTED ALGORITHMS IN DEEP NETWORKS

by

Midshipman 1/C Ryan J. Burmeister
United States Naval Academy
Annapolis, Maryland

(signature)

Certification of Adviser Approval

Professor Gavin W. Taylor
Computer Science Department

(signature)

(date)

Acceptance for the Trident Scholar Committee

Professor Maria J. Schroeder
Associate Director of Midshipman Research

(signature)

(date)

| **REPORT DOCUMENTATION PAGE** | *Form Approved OMB No. 0704-0188* |
|---|---|

| | | |
|---|---|---|
| **1. REPORT DATE** *(DD-MM-YYYY)* 05-11-2016 | **2. REPORT TYPE** | **3. DATES COVERED** *(From - To)* |

| | |
|---|---|
| 4. TITLE AND SUBTITLE<br>Fast, Distributed Algorithms in Deep Networks | **5a. CONTRACT NUMBER** |
| | **5b. GRANT NUMBER** |
| | **5c. PROGRAM ELEMENT NUMBER** |
| **6. AUTHOR(S)**<br>Burmeister, Ryan J. | **5d. PROJECT NUMBER** |
| | **5e. TASK NUMBER** |
| | **5f. WORK UNIT NUMBER** |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** | **8. PERFORMING ORGANIZATION REPORT NUMBER** |
| **9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**<br>U.S. Naval Academy<br>Annapolis, MD 21402 | **10. SPONSOR/MONITOR'S ACRONYM(S)** |
| | **11. SPONSOR/MONITOR'S REPORT NUMBER(S)**<br>Trident Scholar Report no. 446 (2016) |

**12. DISTRIBUTION / AVAILABILITY STATEMENT**

This document has been approved for public release; its distribution is UNLIMITED.

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**

In this project we demonstrate two different approaches to speed up the training of neural nets. First, even before training, we demonstrate an informed way of initializing parameters closer to their final, trained values. Second, we introduce a new training algorithm that scales linearly when parallelized, allowing for substantially decreased training times on large datasets. Neural nets are famously unintuitive, and as such, parameters are typically randomly assigned, then adjusted during training. However, by using a cosine activation function, a layer of neurons can be made to approximate the implicit feature space of a kernel. Therefore, intuition on kernel selection can guide initial parameter assignments even before any data observations. We implement this approach and show that it can greatly speed up training, often approaching the final accuracy after only one training iteration.

Our second contribution was in the application of the ADMM algorithm to neural nets. Conventional gradient based optimization methods for neural nets scale poorly which is difficult to avoid with extremely large datasets. The proposed method avoids many of the conditions that typically make gradient based methods slow, allowing for efficient computation without specialized hardware. Our implementation demonstrates strong scalability with linear speedups even up to thousands of cores. We show that for large problems, our approach can converge faster than GPU-based implementations of standard algorithms.

**15. SUBJECT TERMS**

Machine Learning, Neural Networks, Optimization, Classification

| **16. SECURITY CLASSIFICATION OF:** | | | **17. LIMITATION OF ABSTRACT** | **18. NUMBER OF PAGES** | **19a. NAME OF RESPONSIBLE PERSON** |
|---|---|---|---|---|---|
| **a. REPORT** | **b. ABSTRACT** | **c. THIS PAGE** | | 29 | **19b. TELEPHONE NUMBER** *(include area code)* |

## Abstract

Neural nets frequently outperform other approaches in classification and regression, and have become immensely popular in uses ranging from automatic recognition of handwritten zip code digits to speech recognition on mobile phones. A neural net defines a function with a very large number of tunable parameters that must be learned so the function's output matches the labels this dataset. However, the training of large networks can take days or weeks; as a result, much attention has been given to optimizing networks.

In this project we demonstrate two different approaches to speed up the training of neural nets. First, even before training, we demonstrate an informed way of initializing parameters closer to their final, trained values. Second we introduce a new training algorithm that scales linearly when parallelized, allowing for substantially decreased training times on large datasets.

Neural nets are famously unintuitive, and as such, parameters are typically randomly assigned, then adjusted during training. However, by using a cosine activation function, a layer of neurons can be made to approximate the implicit feature space of a kernel. Therefore, intuition on kernel selection can guide initial parameter assignments even before any data observations. We implement this approach and show that it can greatly speed up training, often approaching the final accuracy after only one training iteration.

Our second contribution was in the application of the ADMM algorithm to neural nets. Conventional gradient based optimization methods for neural nets scale poorly which is difficult to avoid with extremely large datasets. The proposed method avoids many of the conditions that typically make gradient based methods slow, allowing for efficient computation without specialized hardware. Our implementation demonstrates strong scalability with linear speedups even up to thousands of cores. We show that for large problems, our approach can converge faster than GPU-based implementations of standard algorithms.

Keywords: Machine Learning, Neural Networks, Optimization, Classification

**Acknowledgments**

First and foremost, I would like to thank my adviser Professor Gavin Taylor. It is because of your guidance, patience, and motivation that I was able to produce my greatest academic accomplishment to date. Even more than the wealth of knowledge you have passed on, you have become a mentor who was willing to listen through the ups and downs of my 1/C year. I have you to thank for making this project all that it has been. Thank you!

Next, I would like to thank the professors of the Computer Science Department who have been absolutely instrumental in making my four years at the Academy the incredible experience it has been. I came to the Academy knowing nothing about computer science and little about programming, and each of you has helped fuel a passion that I will carry with me the rest of my life. To Professors Albing, Choi, Crabbe, McDowell, and Roche, CDRs Bilzor and Blenkhorn, Maj Pepin, and Capt Sikora, thank you!

Someone who may not have realized how vital she was in making this project a reality is Professor Crainiceanu. Without knowing who you were, you invited me into your office the beginning of my 2/C year and asked what my intentions were for the rest of my time at the Academy. Your suggestion to pursue research and a graduate education has led to incredible opportunities that would not have been possible if it were not for a short conversation. Without you, I do not think I would be sitting here writing this today. Thank you!

To Professor Schroeder and the Trident Committee, thank you for your willingness to support me in my endeavor this year. Professor Schroeder, your devotion to the research and Trident program has been incredible to witness. Most importantly though, your emails were always a “gentle” reminder of everything I had to complete. Thank you all!

Finally, to my fianceé and future wife Hannah, thank you for supporting me through all of the late nights and work-filled weekends. Your willingness to put up with the post midnight phone calls just to say goodnight was more a regular occurrence than I would like to admit, but your support was a constant. Thank you!

## Contents

## List of Figures

# 1 Introduction

What is in an image? Is the picture taken outdoors? Is there a person within the image? How about a dog? Is there grass or was it taken in a desert, or indoors? Humans are naturally capable of distinguishing between all sorts of objects without thinking. For instance, it is easy to understand that the rolling chair the author is sitting on right now and the four-legged chair the author will sit on tonight for dinner are both chairs while having very different properties and appearances. How is it that we are so aptly able to make generalizations about objects of similar type? There must be some understanding of a chair that everyone has, but how do we quantify that understanding? Could a computer learn to make these same generalizations and learn what a chair is?

The problem mentioned above is known as classification. In classification, a machine is given a number of labeled examples (for example, pictures with dogs and pictures without dogs), and learns to distinguish between the different classes so that when shown a new, unlabeled data point, it identifies it accurately. One way computers classify data is using a mathematical model with many adjustable parameters known as a neural network. This model is trained by incrementally adjusting these parameters until the model classifies data points with known classification, the training set, successfully. It is assumed that parameter values which correctly classify labeled examples will also correctly classify new, unlabeled examples known as the test set. A closely related problem is that of regression. In regression, rather than than being placed in one of several discrete classes, the data point must be placed correctly on a continuous number line.

Neural nets are a popular method to classification and regression as they have recently begun to frequently outperform other approaches. However, training of large networks can take a substantial amount of time (on the scale of days or weeks)[16], as they often require the tuning of thousands of parameters. As a result, substantial attention has been given to optimization methods for training networks [10, 20, 24, 34]. The goal of this project is to build upon current methods to make training faster.

## 1.1 Background

Neural nets are networks of interconnected units called neurons, where each neuron consists of input features, weights, and a nonlinear activation function (Figure 1). Features are numerical values describing a data point; for example, the features of an image may be its pixel values. Together the features of a data point are known as its feature set.

In each neuron, the features are linearly combined using a set of weights; this combination is fed into the activation function, which produces the output of the neuron. An individual neuron (i.e. perceptron) can itself be used as a binary classifier [29]. One possible activation which makes sense in this case is the step function (Figure 2a). Here the combination of weights and features are a continuous input which get mapped to either 0 or 1, the classes. However, to enable the learning process, it is desirable for the function to be differentiable, leading to several useful approximations. A perceptron classifier using a logistic sigmoid (Figure 2c) can be interpreted as returning the probability of a data point belonging to class 1 (this type of perceptron classifier is also known as logistic regression). This property is lost in more complex settings, and is more computationally expensive than the hyperbolic

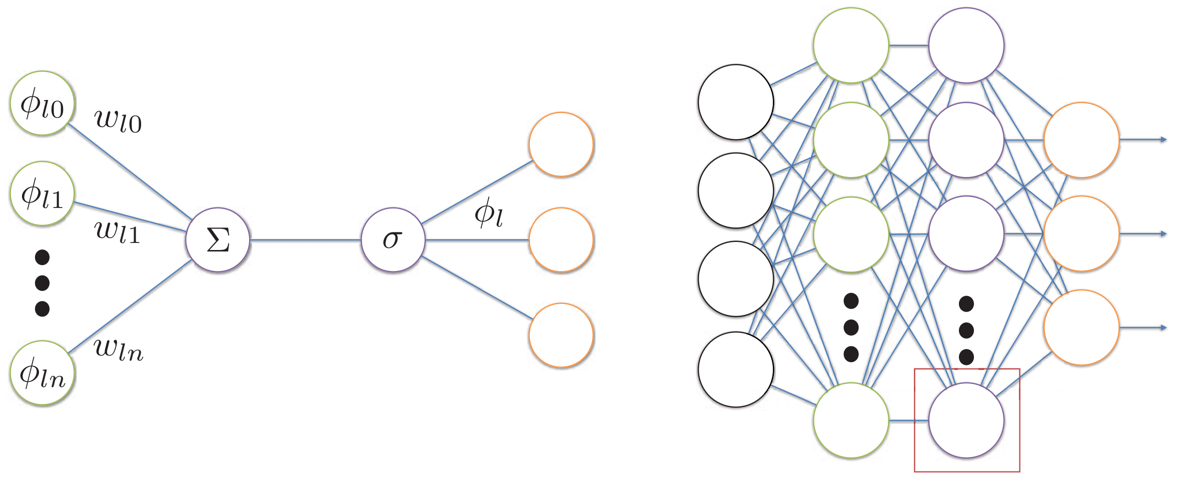


Figure 1: The left image displays a single neuron of a neural network. The input features $\phi_{l-1}$ are weighted by parameters $w_l$, and summed. The output is then fed through an activation function $\sigma(\cdot)$, producing the output $\phi_l$. The image on the right depicts a neural network with two hidden layers and one output layer. The input to any neuron is the output from the previous layer or original features if the first layer. This pattern continues until the final output of the network.

tangent (Figure 2b) [19]. Recently, the ReLU activation (Figure 2d) has become a popular choice due to its computational simplicity [13].

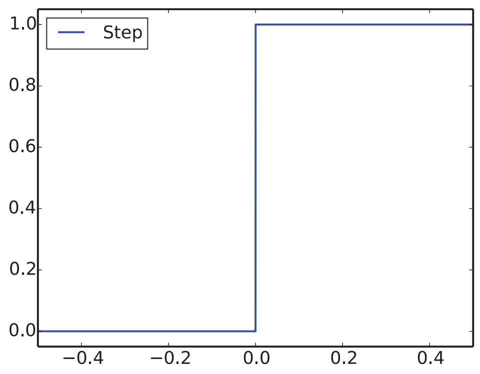
In the most common arrangement, neurons are arranged in layers where the input to each neuron in the layer is the output from all the neurons in the previous layer. The network functions by propagating the input to the first layer, the data's feature set, through the network to the final layer where the output, the supposed classification, is produced. This arrangement allows for highly nonlinear and complex classification and regression models capable of approximating nearly any function (for example, [8, 11, 15]).

## 1.2 Mathematical Formulation of Neural Nets

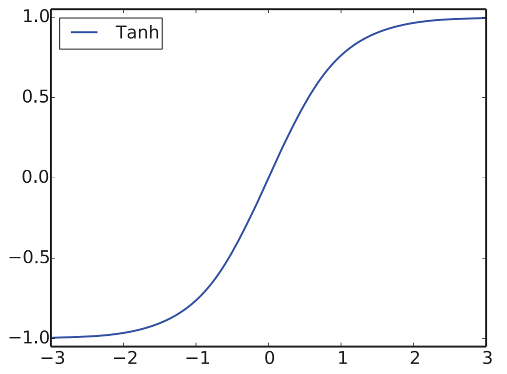
Let $\phi_0$ represent a $k_1 \times n$ matrix of all $n$ training points, each of which is described by $k_1$ features; more generally, let $\phi_{l-1}$ be the $k_l \times n$ matrix representing the input matrix of all data to the $l$th layer and the output of the $(l-1)$th layer. Additionally, let $w_{li}$ represent the $1 \times k_l$ weights of the $i$th neuron in layer $l$, and let $W_l$ represent the $m_l \times k_l$ concatenation of all weight vectors for the $m_l$ neurons in layer $l$. Finally, let $\sigma(\cdot)$ be a nonlinear activation function. We can then mathematically represent the entire $L$-layer network concisely with

$$\phi_L = W_L \sigma_{L-1}(\ldots \sigma_2(W_2 \sigma_1(W_1 \phi_0)) \ldots). \tag{1}$$

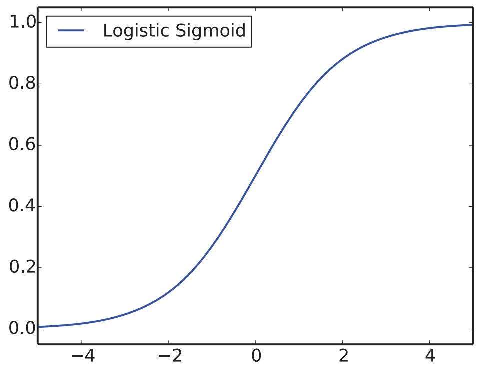
$\phi_L$ is the predicted classification for the $n$ data points described in $\phi_0$. On an output node, the resulting value may be interpreted as a likelihood (in a colloquial sense) of the data being a member of a particular class. Note, the amount of data stored and transmitted during the learning process not only depends on the number of data points but also on the number of features for each data point and size of the network.



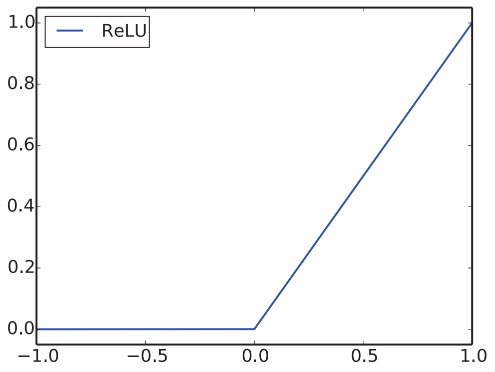
(a) Step



(b) Hyperbolic Tangent



(c) Logistic Sigmoid



(d) ReLU

Figure 2: These graphs depict activation functions. The step function is approximated by both the hyperbolic tangent and logistic sigmoid functions and is useful as an example for binary classification. The ReLU function is currently a popular choice due to its computational simplicity.

## 1.3 Goal of Training

The only way to change the output of the network is by changing the weights - unfortunately, neural nets describe extremely complex functions, resulting in little intuition or theory describing appropriate weight values [22, 16, 12, 14, 31]. Therefore, weights are randomly initialized and iteratively updated during training until the network output closely matches the known classification of the labeled dataset (for example, [19]). The resulting weights are then mathematically useful, though not meaningful to humans. We now describe this training process in more detail.

The goal of training is for the output of the net $\phi_L$ to be as close as possible to the known output, $t$ ($t \approx \phi_L$). The difference can be measured by $\ell(\cdot, \cdot)$, a loss function describing the distance between two vectors (for example, the $L_2$ distance). We can describe training with the optimization problem

$$\underset{W_{\{0:L-1\}}}{\text{minimize}} \quad \ell\left(\phi_L, t\right). \tag{2}$$

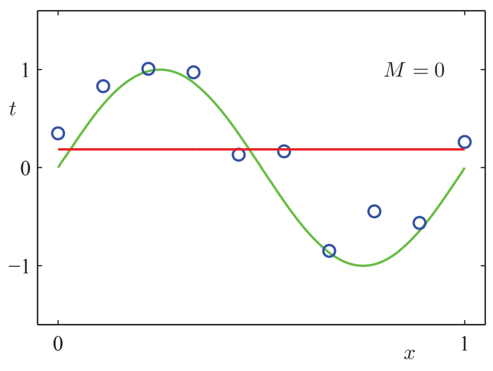
Training can occur in a number of ways, but the goal is the same for each. The weights are iteratively updated in the direction which minimizes the objective function. Training is complete when (2) converges, or stated alternatively, when the difference between $t$ and $\phi_L$ can no longer be considerably improved.

## 1.4 The Necessity of Data

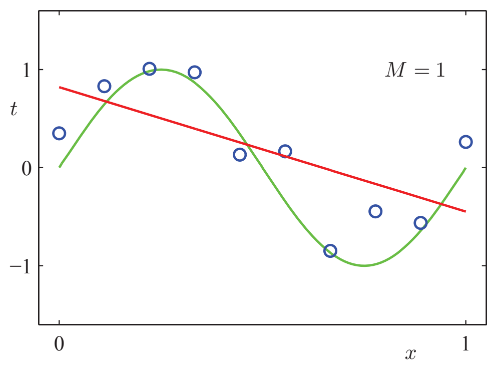
While training, the goal is to create a model which can accurately predict labels for new points not seen during the training phase. One problem that can arise during training is an overly expressive model. In this scenario, the model nearly perfectly fits the training data but fails to accurately perform classification on new unseen data points. This is known as overfitting and is a common problem in many machine learning applications.

In a neural network, there often exists thousands tunable parameters. However, if the number of parameters is too great for the given dataset, then overfitting can occur. In figure 3, different regression models are produced using a variable number of parameters. The model more accurately fits the training data as more parameters are introduced; as the number of parameters increases, this begins to come at the expense of accurate predictions. New y-values would be predicted based on the red line which for large numbers of parameters is clearly an inaccurate representation of the data's source.

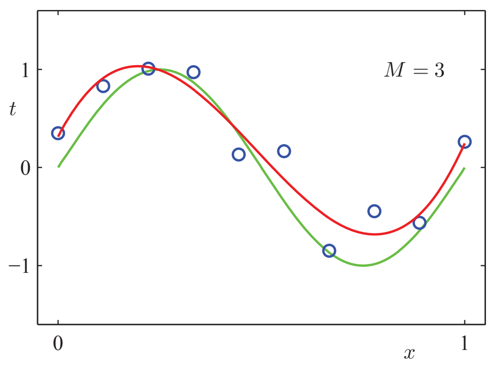
One way to combat overfitting is through the introduction of more data. Even with a large number of parameters, a significant amount of data ensures that overfitting is unlikely and that the model fits the distribution from which the data was sampled, not the observed training points. The size of datasets used for difficult classification and regression problems is often in the range of several TB of data; though this effectively overcomes the overfitting problem, it makes efficient computation much more difficult. One way to combat the efficiency problem is through expensive, special-purpose hardware, such as GPUs manufactured specifically for neural-net computation. In contrast, our approach is to improve the training algorithms to better fit existing hardware.



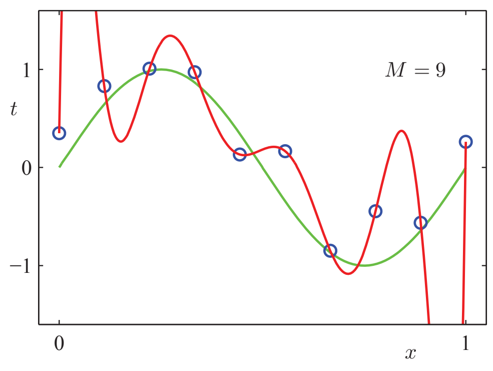
(a) 1 Parameter



(b) 2 Parameters



(c) 4 Parameters



(d) 10 Parameters

Figure 3: Each image displays a representation of a linear model used to fit the data with varying numbers of nonlinear monomial features [3], where $M$ indicates the number of such features aside from the bias feature. So, the total number of unknowns in each image is $M + 1$. The blue circles are noisy observations sampled from the green line. The red line is the model's prediction of what the curve should be. Note that though the model is a linear combination of features, the features are nonlinear, resulting in a nonlinear prediction line.

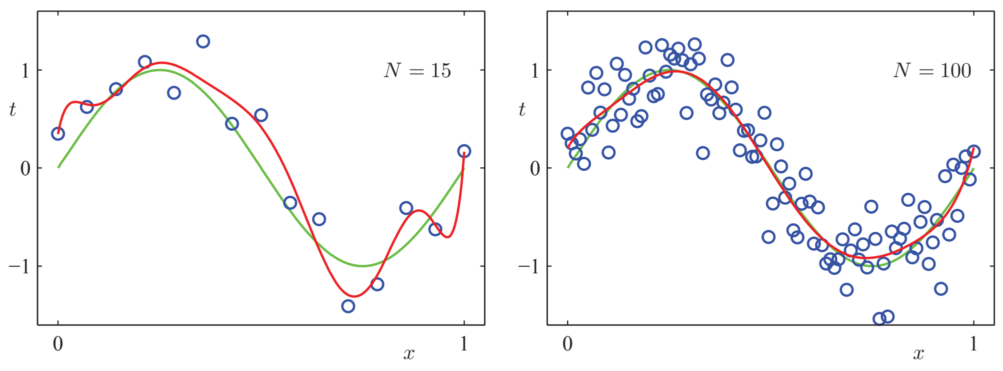


Figure 4: In these images [3], the usefulness of data is demonstrated. The green line again represents the sine curve with the blue dots representing data taken from a noisy source. The red line is the model's prediction curve when there are 10 features (i.e. $M = 9$) and on an increased number of data points (i.e. 10 data points in figure 3d compared to 15 and 100 in the above). The left image contains more features and data than the figure 3d, and although overfitting still occurs, the effect has been limited by an increase in data. The image on the right shows how it becomes more difficult to overfit with an increase in the amount of data. With 100 data points, an accurate model is produced even with 10 features.

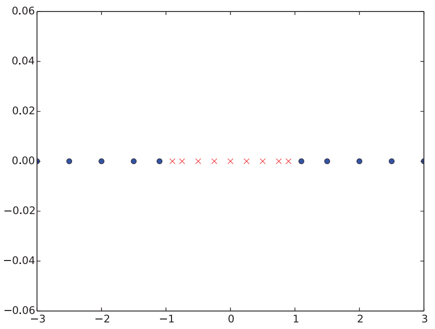
### 1.5 Evaluating Success

In experiments, labeled data is typically split into two disjoint sets, a large training set, and a smaller testing set. The effectiveness of the model is typically evaluated by measuring the test set accuracy following the training of a network on the training set. The bar for "success" is heavily dependent on the difficulty of the problem. Researchers have been able to demonstrate remarkable accuracy on easy problems [33, 24], whereas on hard problems, the state-of-the-art accuracy threshold is much lower [31]. Therefore, the success of implementations must be evaluated based on historical performance on the given dataset. For the purposes of this project, our focus was on reaching state of the art accuracy while reducing the amount of time spent on training.

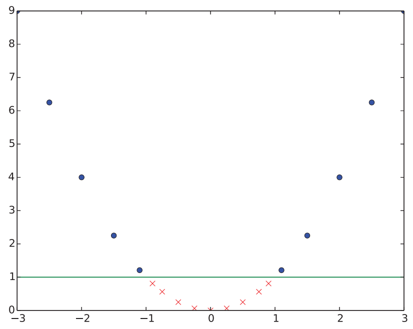
## 2 Kernel Guided Weight Initialization

We introduce two techniques to speed up the process by which weights converge on their final values: informed weight initialization and faster general purpose algorithms for weight training. In this section, we focus on the first approach.

As the purpose of training is to obtain the optimal set of weights which minimize a cost function, a natural way to increase the rate of training would be to initialize the weights close to their final trained values. However, as weight initialization occurs prior to any data observation by the network, this requires intuition by the creator of the network as to what appropriate weight values should be. Unfortunately, neural nets are extremely unintuitive, often with thousands of tunable parameters interacting in complex ways. Consequently, the state-of-the art approaches simply rely on random initialization. We propose an alternative



(a) Features in 1-dimensional space



(b) Features in higher 2-dimensional space

Figure 5: In these images, it is clear how feature engineering can be important in linear models. In figure 5a, the line of points of two classes is impossible to separate linearly. However, if an additional feature of $x^2$ is added, the points in the higher two dimensional space are linear separable as shown by the green line in 5b. Kernels allow for the use of features in an extremely high dimensional space which likely allows for linear separation of classes.

approach in which weights are initialized with kernel based intuition or a quick hyperparameter search.

Kernels were originally introduced as an alternative to linear models which allow for avoidance of feature set selection difficulties [21, 1], and have a long history in alternatives to neural nets such as support vector machines [4] and Gaussian processes [28]. The intuition leveraged in these approaches is that nearby points are likely to be of similar classification. In order to define what "nearby" is, a kernel function on two points is defined. For example, a common kernel function is a multivariate Gaussian centered around one of the points entered into the function - the closer the points are, the larger the kernel value will be. A "neighborhood" is then defined by the covariate matrix of the Gaussian. If a researcher believes two points need to be quite close for their relationship to be meaningful, she might use a covariate matrix with small values. Because kernelized methods allow for the application of this type of intuition given a particular dataset, we would like to combine the advantages of kernels with neural nets. However, kernels have not traditionally fit into the neural net paradigm.

## 2.1 Introduction to Kernels

Kernelized methods are usually introduced as an alternative to linear models, such as linear or logistic regression. In linear models, the output is determined by a linear combination of features, the weights for which are learned from the training data (recall that a single neuron, common called a perceptron, is itself a linear model). In these models, the selection of useful features is extremely important so the classification or regression is performed in an advantageous space, as is illustrated in Figure 5; this choice of feature set is extremely important, and frequently difficult (for example, [32]).

Kernelized methods are derived from the dual representation of the linear model. This dual representation allows for the problem to be defined in a way that does not require the introduction of an appropriate feature set which is often difficult to come up with. By working in terms of the dual, implicit feature spaces of high or even infinite dimensionality can be utilized. This expressive space can ease classification as points of similar classification are more likely to be separable from others in a higher dimensional setting than the original low dimensional space.

A kernel function is defined as a similarity function,

$$k(\mathbf{x}, \mathbf{y}) = \varphi(\mathbf{x})^T \varphi(\mathbf{y}), \tag{3}$$

where $\mathbf{x}$ and $\mathbf{y}$ are two training points, $k$ is the kernel function which defines a similarity between $\mathbf{x}$ and $\mathbf{y}$, and $\varphi$ maps the inputs to a higher dimensional space.

## 2.2 Incorporating Kernels into Neural Nets

To incorporate kernels into neural nets, we approximate the kernel function through a linear combination of Fourier bases. Rahimi and Recht [26] showed shift-invariant kernels can be mapped to low-dimensional Euclidean inner product spaces through the use of a random feature map

$$k(\mathbf{x}, \mathbf{y}) = \varphi(\mathbf{x})^T \varphi(\mathbf{y}) \approx \mathrm{z}(\mathbf{x})^T \mathrm{z}(\mathbf{y}), \tag{4}$$

where $\mathbf{x}$, $\mathbf{y} \in R^d$, $R^d$ is the original feature space, $\varphi$ is the implicit higher dimensional feature space defined by the kernel, and z is the randomized feature map.

To calculate our feature map, we first evaluate the Fourier Transform $p(\omega)$ of the desired kernel function where $\omega \in R^d$. The kernel can then be evaluated in expectation

$$k(\mathbf{x}, \mathbf{y}) = E[z_\omega(\mathbf{x}) z_\omega(\mathbf{y})] \tag{5}$$

$$z_\omega(\mathbf{x}) = \sqrt{2}\cos(\omega^T \mathbf{x} + b), \tag{6}$$

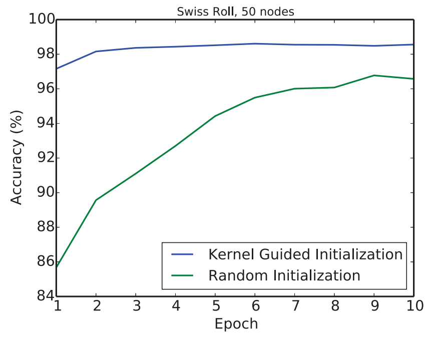
where $\omega \sim p(\omega)$ and $b \sim [0, 2\pi]$ [26]. Therefore, we can define the feature set

$$\mathrm{z}(\mathbf{x}) \equiv \sqrt{\frac{2}{D}}[\cos(\omega_1^T \mathbf{x} + b_1) \;\ldots\; \cos(\omega_D^T \mathbf{x} + b_D)]^T, \tag{7}$$

where $\omega_i$ and $b_i$ are drawn independently from $p(\omega)$ and $[0, 2\pi]$, respectively. The linear space of this feature set approximates the linear space of the high-dimensional space $\varphi$.

Given this previous work, we propose the introduction of an alternative activation function $\sqrt{\frac{2}{D}}\cos(\cdot)$ to replace the more common logistic sigmoid, hyperbolic tangent, or ReLU activation functions. This now means that we can approximate the linear space implied by a kernel by initializing weights independently for each neuron in a layer, where $\omega \sim p(\omega)$ and $b \sim [0, 2\pi]$. Furthermore, assuming we have chosen a good kernel function, as the data is linearly separable in the linear space of $\varphi$, they are likely to be separable in the linear space of z as well.

This activation function offers two benefits. First, problems well suited to kernelized approaches, such as those that lie on a lower-dimensional manifold, can now be approached



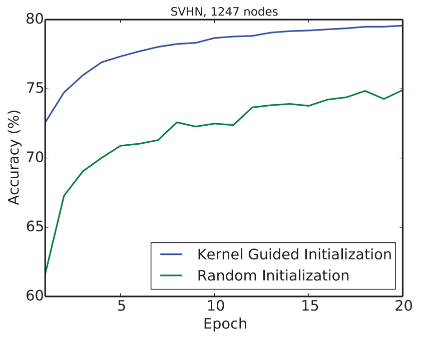


Figure 6: The blue lines are tests run with cosine activation functions and intelligent initialization, while the green lines are run with a traditional hyperbolic tangent activation and random initialization. Both tests were run so that both the hyperbolic tangent and cosine networks were set to an optimal setting. The graph on the left is for the Swiss roll dataset, while the graph on the right is for the Google Streetview dataset.

in the same way by neural nets. Second, because a particular kernel implies the distribution from which to sample the initial weights, the same intuition that frequently eases kernel selection eases weight initialization, ultimately shortening training time. Even absent this intuition, because kernel functions are defined over a small number of parameters, a quick, small-dimensional parameter search is highly effective. For instance, in symmetric Gaussian kernels, the search is simply over standard deviation, a one-dimensional space. All that needs to be done is to test values of the standard deviation to determine which value performs the best for the particular problem at hand. Given a sense of what kernel to use given a particular dataset, our method provides a way to set up the weights of a layer of a neural net such that it approximates this kernel function and requires little training.

## 2.3 Experimental Results

To test the effectiveness of the kernel-guided initialization, we evaluated our method on a synthetic Swiss Roll and the real-world Google Streetview datasets. To do this, the neural net package Torch was used to allow for highly-optimized standard optimization approaches.

The Swiss Roll dataset is a synthetic dataset consisting of 160,000 data points, 120,000 of which were used for training with the remaining being used for testing. The dataset was generated by creating a 2-D ribbon in which opposing corners of the ribbon were of the same class. This ribbon was then cast to a 3-D space to create the Swiss Roll shape (Figure 7). The dataset was appropriate for this task as the data could not be linearly separated in the original feature space which consisted of its $x$-, $y$-, and $z$-coordinates.

The Google Streetview [23] dataset consists of images of house numbers taken from the Google Streetview car. Each data point consisted of a cropped image of a single digit which was human-labeled. The goal was to determine which digit appeared in each of the images, resulting in 10 classes; random chance is therefore roughly 10% accuracy. The dataset contained 99,289 data points, 73,257 of which were used for training with the remaining used for testing. For initial features, we used 684 Histogram of Oriented Gradients (HOG) [9] features of the original images. The kernel we used was a symmetric Gaussian kernel;

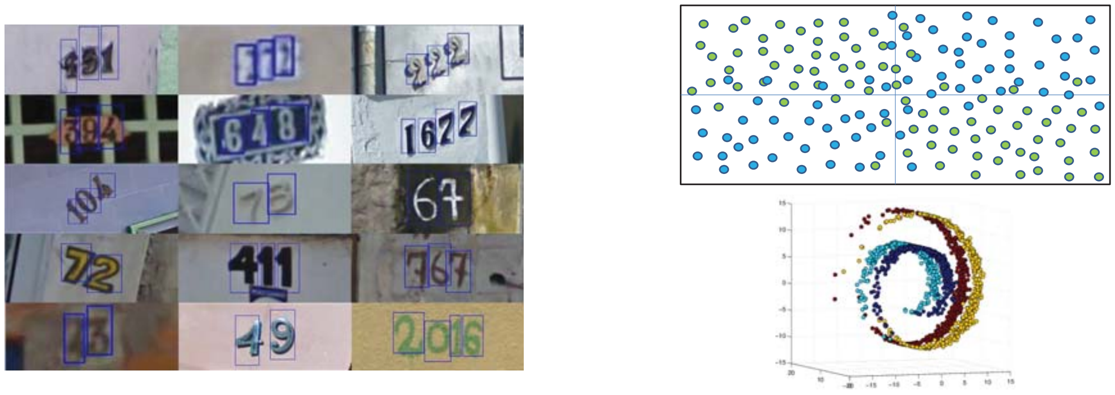


Figure 7: The left image displays examples of images taken from the Google Streetview dataset [23]. The images used for training and testing were contained entirely within the blue rectangles. The image on the right displays 2-D and 3-D representations of the Swiss roll dataset.

we ran a preliminary hyperparameter search to discover the optimal standard deviation for the distribution to be used when initializing the weights. The search was performed over a single iteration of training with a number of different standard deviation values. The value selected was the standard deviation which resulted in the highest test set accuracy.

The graphs (Figure 7) indicate the predictive accuracy of the neural net on the test set as a function of training epochs. In both instances, a single hidden layer was utilized; the random initialization trials used a hyperbolic tangent activation function. Quick hyperparameter searches were performed to ensure the networks were set with ideal initial parameters to ensure convergence was reached as quickly as possible. A grid search for the optimal learning rate, and weight decay was done for each of the functions independently. An additional search was done for the number of hidden nodes that was most effective for the hyperbolic tangent activation functions; this architecture was then used by both approaches.

The graphs' x-axes depict the number of epochs training occurred, where an epoch is defined to be one update of the weights after training on a random subset of the data; therefore, an increase in epochs results in generally improved accuracy. Initializing weights in an intelligent manner gave the neural net with a cosine activation function far better initial accuracy which led to substantially faster convergence and better overall performance even after substantial training.

With the cosine activation function, we were able to obtain results better than the current state of the art for a neural network with a single hidden layer. Furthermore, we were able to show that while obtaining better results, the network converged substantially faster to an optimal set of weights.

## 3 Improved General Use Algorithm

In this section, we introduce a new optimization method for training neural networks, which is appropriate for very large datasets distributed across multiple machines. The manner in

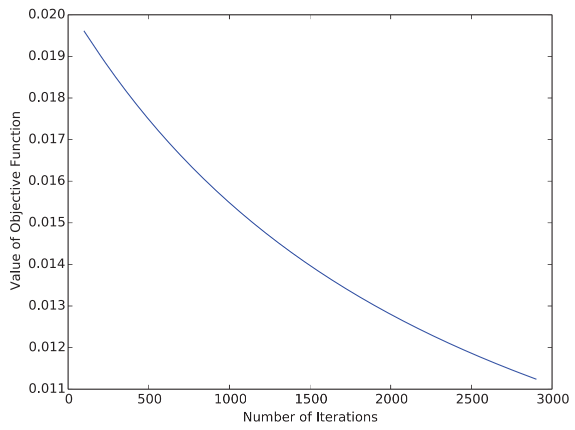


Figure 8: During gradient descent, the goal is to minimize an objective function by updating the weights. The y-axis represents the value of the objective function while the x-axis signifies the number of iterations. During each iteration, an update to the weights occurs using gradient descent.

which weights are updated is traditionally through gradient methods [5, 18, 30, 6, 35]. To motivate our new method, we first discuss these gradient methods and their weaknesses.

## 3.1 Gradient Descent

Gradient descent is a method by which an objective function $f(w)$ can be minimized by taking the gradient with respect to parameters $w$ and updating the parameter values with the update rule

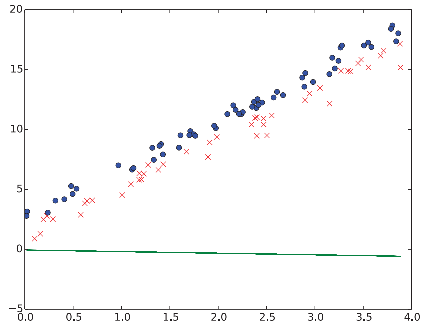
$$w_{t+1} = w_t + \eta \nabla f(w), \tag{8}$$

where $\eta$ is the learning rate. $\eta$ plays a substantial role in performance by controlling the size of the update, and usually requires substantial experimental tuning.

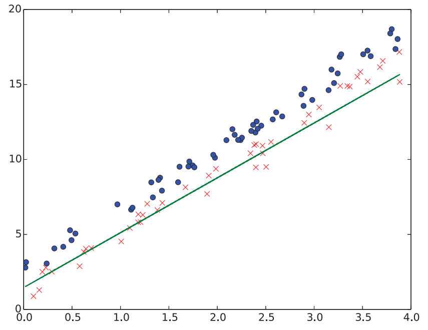
We demonstrate gradient descent with linear classification in a toy problem in Figures 8 and 9. As shown in Figure 8, initially, the objective function has a large value which signifies a large difference between the predicted and output classification values. As the difference between labeled and predicted points approaches zero our error decreases and our decision boundary improves. Figure 9 shows the classification boundary at intermediate points in the optimization.

### 3.1.1 Gradient Saturation

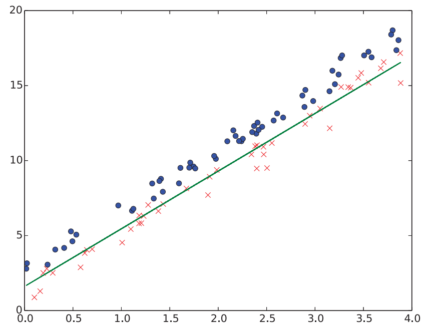
Gradient descent depends upon an assumption that away from an optimum, the gradient will be large enough to be informative, but when close to an optimum, the gradient will be small enough to limit the change in weight updates. When the assumption is not met, gradient descent can perform very poorly. For this reason, optimization problems with saddle points, or points of very small gradient that are nonetheless not optima, can be poor fits for gradient descent. This is because a large number of iterations, each changing the parameters a very small amount, would be required to move past the saddle point.



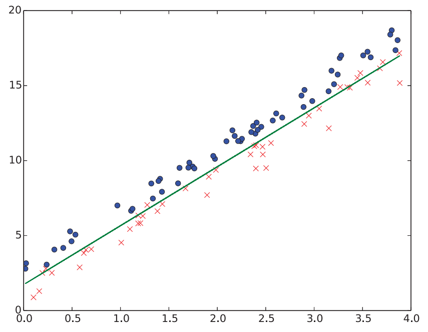
(a) 0 iterations



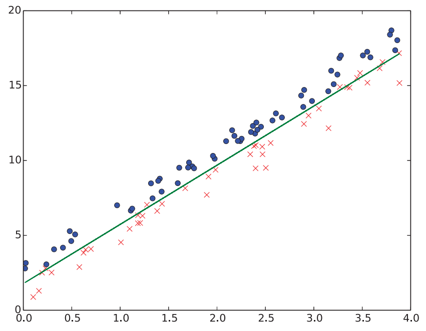
(b) 500 iterations



(c) 1000 iterations



(d) 2000 iterations



(e) 3000 iterations

Figure 9: Training is demonstrated as the production of a decision boundary using linear classification with three features $(x, y, 1)$. The data was made by generating points from $y = 4x + 2 + \epsilon$, where $x$ is is a random sampling of points in the range $[0, 4)$ and $\epsilon$ was drawn from a normal distribution with mean 0 and standard deviation 1 to represent a noisy system. If $\epsilon$ was positive, then the point was classified as a blue circle, but if negative, then it was labeled as a red "x". Gradient descent is applied to update the weights after each iteration. Examples after 0, 500, 1000, 2000, and 3000 iterations are shown.

We illustrate the slow rate of progress that can result in figure 9. As the solution approaches an optimum, the gradient decreases rapidly, resulting in much less progress per iteration. It is important to note this illustration is not a saddle point, and so differs in that sense from the problem with neural nets. Nonetheless, the figure demonstrates the amount of work necessary to overcome an uninformative gradient.

Unfortunately, neural nets create nonconvex spaces with many saddle points and other areas where the gradient is uninformative. Common activation functions such as logistic sigmoids or hyperbolic tangents feature large areas of near-zero slope, inducing saddle points in the overall objective function (2). This causes weight updates to be very small and unproductive until past the saddle point; this is known as gradient saturation and is the motivating issue behind our new gradient-free optimization algorithm of section 3.

### 3.1.2 Poor Scalability

In addition to computational problems associated with gradients, gradient based methods also demonstrate poor scalability as the amount of data substantially increases and parallelism becomes required [27, 7]. One of the major reasons neural nets and deep learning have been so successful is the increasing amount of data available. As the amount of data available for training increases, the better the model typically performs. However, while updating gradient information in serial performs very well, it lacks properties which allow it to scale well.

Gradient descent scales poorly as it relies on a large number of inexpensive minimization steps with required communication between all nodes after each iteration; this frequent communication quickly becomes expensive. Consequently, the utilization of highly specialized GPUs, where memory is shared, has become an effective way to combat communication costs. GPUs allow for thousands of threads to simultaneously work on small batches of data with minimal communication overhead. While this hardware is remarkable, it remains highly expensive and specialized, and can still be overwhelmed by a large enough problem, requiring communication.

Conversely, due to higher communication costs, CPU methods are best when they rely on a small number of expensive minimization steps. The cost of minimization can then be split among many nodes, allowing for the cost of communication to be amortized over a substantial amount of computation.

## 3.2 ADMM for Neural Networks

We propose a CPU-based solution to address the problems mentioned previously by splitting the objective function of each layer of a network into several terms. The Alternating Direction Method of Multipliers (ADMM) method allows for the computation of each term separately and in closed form, without gradient information. This changes the optimization problem from iterations over a single difficult problem (updating weights based on gradients) into a series of very easy problems. Additionally, several of these problems are easily parallelized, leading to linear scaling across the number of compute cores, which has not been achieved by more traditional approaches.

For reference, we reprint the formulation (1) and the minimization problem (2).

$$\phi_L = W_L \sigma_{L-1}(\ldots \sigma_2(W_2 \sigma_1(W_1 \phi_0)) \ldots).$$

$$\underset{W_{\{0:L-1\}}}{\text{minimize}} \quad \ell\left(\phi_L, t\right).$$

The intuition behind the ADMM method is to decouple the weights from the activations. Instead of feeding the product $W_l \phi_{l-1}$ directly into the activation function $\sigma_l$, the result is stored in a new variable $z_l = W_l \phi_{l-1}$. The output for a layer in the network can the be represented as a vector of activations $\phi_l = \sigma_l\left(z_l\right)$. This reorganizes the optimization problem as

$$\begin{aligned} \underset{W_{\{l\}}, \phi_{\{l\}}, z_{\{l\}}}{\text{minimize}} \quad & \ell\left(z_L, t\right) \\ \text{subject to} \quad & z_l = W_l \phi_{l-1} \text{ for } l = 1, 2, \ldots, L \\ & \phi_l = \sigma_l\left(z_l\right) \text{ for } l = 1, 2, \ldots, L-1. \end{aligned} \tag{9}$$

Note that solving (9) is equivalent to solving (2). However, rather than solving (9) directly, an $L_2$ term is added in order to relax the constraints. The unconstrained problem

$$\underset{W_{\{l\}}, \phi_{\{l\}}, z_{\{l\}}}{\text{minimize}} \quad \ell\left(z_L, t\right) + \beta_L \left\| z_L - W_L \phi_{L-1} \right\|_2 + \sum_{l=1}^{L-1} \left[ \gamma_l \left\| \phi_l - \sigma_l\left(z_l\right) \right\|_2 + \beta_l \left\| z_l - W_l \phi_{l-1} \right\|_2 \right] \tag{10}$$

can then be attacked, where $\gamma_l$ and $\beta_l$ are constants which control the weight of each constraint. However, the constraints in (10) are only approximately enforced. In order to exactly enforce the constraints in the last layer and induce stability, a Lagrange multiplier term, $\lambda$ is added to the final layer, which yields

$$\begin{aligned} \underset{W_{\{l\}}, \phi_{\{l\}}, z_{\{l\}}, \lambda}{\text{minimize}} \quad & \ell\left(z_L, t\right) + \left\langle z_L, \lambda \right\rangle \\ & + \beta_L \left\| z_L - W_L \phi_{L-1} \right\|_2 + \sum_{l=1}^{L-1} \left[ \gamma_l \left\| \phi_l - \sigma_l\left(z_l\right) \right\|_2 + \beta_l \left\| z_l - W_l \phi_{l-1} \right\|_2 \right]. \end{aligned} \tag{11}$$

The split formulation (11) is designed so that it can be easily minimized through a series of sub-steps, each of which can be solved for in closed-form. The ADMM algorithm updates one set of variables at a time - either $W_l$, $\phi_l$, or $z_l$ - while holding the others constant.

### 3.2.1 Minimization sub-steps

This section focuses on the update procedure for each variable in (11). The algorithm proceeds by minimizing for $W_l$, $\phi_l$, and $z_l$, and then updating the Lagrange multipliers $\lambda$.

The minimization of (11) with respect to $W_l$ is first considered. For each layer $l$, the optimal solution minimizes $\left\| z_l - W_l \phi_{l-1} \right\|_2$. This is a least-squares problem whose solution is given by $W_l \leftarrow z_l \phi_{l-1}^{\dagger}$.[1]

[1] † represents the pseudoinverse

To update the activations, a process similar to the one done for the weights is performed. However, the activation matrix $\phi_l$ appears in two penalty terms in (11). Therefore, the minimization occurs over $\beta_l \left\| z_{l+1} - W_{l+1}\phi_l \right\|_2 + \gamma_l \left\| \phi_l - \sigma_l \left( z_l \right) \right\|_2$ for all $\phi_l$ while all other variables are held fixed. The new value of $\phi_l$ is then given by

$$\phi_l \leftarrow \left( \beta_{l+1} W_{l+1}^T W_{l+1} + \gamma_l I \right)^{-1} \left( \beta_{l+1} W_{l+1}^T z_{l+1} + \gamma_l \sigma_l \left( z_l \right) \right). \tag{12}$$

The update to $z_l$ requires the update

$$z_l \leftarrow \arg\min_z \gamma_l \left\| \phi_l - \sigma_l \left( z \right) \right\|_2 + \beta_l \left\| z - W_l \phi_{l-1} \right\|_2. \tag{13}$$

This problem is non-convex and non-quadratic due to the non-linear term $\sigma$. Fortunately, because the non-linearity $\sigma$ works entry-wise on its argument, the entries in $z_l$ are de-coupled. Furthermore, (13) is particularly easy to solve when $\sigma$ is piecewise linear as it can then be solved for in closed form. For instance, common piecewise linear choices for $\sigma$ include rectified linear units (ReLUs) and non-differentiable sigmoid functions given by

$$\sigma_{relu}(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}, \sigma_{sig}(x) = \begin{cases} 1, & \text{if } x \geq 1 \\ x, & \text{if } 0 < x < 1 \\ 0, & \text{otherwise} \end{cases}.$$

For such choices of $\sigma$, the minimizer of (13) can be computed simple if-then logic. For more sophisticated choices of activation functions, such as a smooth sigmoid as discussed in section 3.1, the sub-step can be solved with a lookup table of precomputed solutions because each 1-dimensional problem only depends on two inputs.

After the minimization steps for $W_l$, $\phi_l$, and $\sigma_l$, the Lagrange update is given by

$$\lambda \leftarrow \lambda + \beta_L (z_L - W_L \phi_{L-1}). \tag{14}$$

Together these updates contribute to produce algorithm (1).

**Algorithm 1** ADMM for Neural Nets

```
**Input:** training features $\{\phi_0\}$, and labels $\{y\}$,
**Initialize:** allocate $\{\phi_l\}_{l=1}^{L=1}$, $\{z_l\}_{l=1}^{L}$, and $\lambda$
**repeat**
**for** $l = 1, 2, \cdots, L-1$ **do**
    $W_l \leftarrow z_l \phi_{l-1}^{\dagger}$
    $\phi_l \leftarrow (\beta_{l+1} W_{l+1}^T W_{l+1} + \gamma_l I)^{-1} (\beta_{l+1} W_{l+1}^T z_{l+1} + \gamma_l \sigma_l(z_l))$
    $z_l \leftarrow \arg\min_z \gamma_l \|\phi_l - \sigma_l(z)\|_2 + \beta_l \|z_l - W_l \phi_{l-1}\|_2$
**end for**
$W_L \leftarrow z_L \phi_{L-1}^{\dagger}$
$z_L \leftarrow \arg\min_z \ell(z, y) + \langle z, \lambda \rangle + \beta_L \|z - W_L \phi_{L-1}\|_2$
$\lambda \leftarrow \lambda + \beta_L (z_L - W_L \phi_{L-1})$
**until** converged
```

### 3.2.2 Distributed Implementation

The ADMM method is scaled using a data parallelization strategy in which different nodes store different activations, outputs, and Lagrange multipliers corresponding to different subsets of data. Consider $N$ worker nodes across which the ADMM algorithm can be distributed. For each layer, the data is broken into chunks for which each node can operate on. The activation matrix, for example, is broken into subsets as $\phi_l = [\phi_l^1, \phi_l^2, \ldots, \phi_l^N]$. The output matrix $z_l$ and Lagrange multipliers $\lambda$ decompose similarly.

The optimization sub-steps for updating $\phi_l$ and $z_l$ do not require any communication and parallelize trivially. The weight matrix update requires the computation of pseudo-inverses and products involving the matrices $\phi_l$ and $z_l$. This can be done effectively using transpose reduction strategies that reduce the dimensionality of matrices before they are transmitted to a central node.

The weight update has the form $W_l \leftarrow z_l \phi_{l-1}^{\dagger}$, where $\phi_{l-1}^{\dagger} = \phi_{l-1}^T(\phi_{l-1}\phi_{l-1}^T)^{-1}$. This formulation allows the $W$ update to decompose across nodes as

$$W_l \leftarrow \left( \sum_{n=1}^{N} z_l^n (\phi_{l-1}^n)^T \right) \left( \sum_{n=1}^{N} \phi_{l-1}^n (\phi_{l-1}^n)^T \right)^{-1}.$$

Each of the individual products $z_l^n(\phi_{l-1}^n)^T$ and $\phi_{l-1}^n(\phi_{l-1}^n)^T$ are computed separately on each node. The resultant matrices can then be summed utilizing a single reduce operation to the central server node. Because the number of features is frequently much smaller than the number of data points, the matrix $\phi_{l-1}^n(\phi_{l-1}^n)^T$ is significantly smaller than $\phi_{l-1}^n$ itself. As the transmission of data to nodes is often a bottleneck, reducing the amount of information required to be transmitted can substantially reduce the run time. Once the products on each node have been formed and reduced onto a central server, the central node computes the inverse of $\phi_{l-1}\phi_{l-1}^T$, updates $W_l$, and then broadcasts the result to the worker nodes.

The update (12) decomposes trivially across workers, with each worker computing

$$\phi_l^n \leftarrow \left(\beta_{l+1} W_{l+1}^T W_{l+1} + \gamma_l I\right)^{-1} \left(\beta_{l+1} W_{l+1}^T z_{l+1}^n + \gamma_l \phi_l\left(z_l^n\right)\right).$$

Each node maintains a full representation of the entire weight matrix, and can formulate its own local copy of the matrix inverse $(\beta_{l+1} W_{l+1}^T W_{l+1} + \gamma_l I)^{-1}$.

Like the update to the activations, the update for $z_l$ trivially parallelizes where each work node solves

$$z_l^n \leftarrow \arg\min_z \gamma \|\phi_l^n - \sigma_l(z)\|_2 + \beta \|z - W_l \phi_{l-1}^n\|_2. \tag{15}$$

Each worker node computes $W_l \phi_{l-1}^n$ using local data, and then updates each of the (decoupled) entries in $z_l^n$ by solving a 1-dimensional problem in closed form.

The Lagrange multiplier update also trivially splits across nodes, with worker $n$ computing

$$\lambda^n \leftarrow \lambda^n + \beta(z_L^n - W_L \phi_{L-1}^n) \tag{16}$$

using only local data.

## 3.3 Experiments

For this section, we compare the results of our ADMM method to frequently-used gradient based approaches, including stochastic gradient descent (SGD), conjugate gradients, and L-BFGS on two benchmark classification tasks. On these two tasks, we illustrate the success of ADMM using two graphs. The first illustration shows the linear scaling of ADMM by varying the number of cores computing in parallel and clocking the compute time to a predetermined accuracy threshold. The second illustration demonstrates the test accuracy as a function of time for each of the optimization methods.

For the purpose of the testing, the ADMM method was compared against the gradient-based methods of stochastic gradient descent (SGD), conjugate gradients, and L-BFGS. In section 3.1, a simple gradient descent method was introduced for convex optimization problems. However, when working with many parameters, the space is nearly always nonconvex and is scattered with many local optima and saddle points which can make learning difficult. Furthermore, gradient descent is often limited by the learning rate. If the learning rate is too large, the gradients diverge. If it is too small, optimization is slow. The alternative gradient methods to standard gradient descent allow for quicker convergence and the ability to alleviate some of the problems typically associated with nonconvex spaces.

Each of the gradient methods mentioned were tested using open source implementations in Lua with Torch. This allowed for testing using these standard approaches to occur on a GPU where they are fastest. In contrast, the ADMM method was run on a variable number of CPUs. Because ADMM is sufficiently different from other optimization approaches, Torch or other neural net libraries were not options, causing us to implement neural nets from the ground up in Python.

The ADMM method was tested on a Cray XC30 supercomputer with Ivy Bridge processors with communication between cores done using MPI. SGD, conjugate gradients, and L-BFGS were implemented on NVIDIA Tesla K40 GPUs. Prior to testing, each of these methods underwent a thorough grid hyperparameter search to determine the optimal setting of running conditions which would produce the best results. In each of the graphs below, the time graph only includes the time spent optimizing, not the time to load the data or set up the network.

Experiments were run on two datasets. The first was the Google Streetview House Numbers (SVHN) dataset previously mentioned. Histogram of Gradients (HOG) features were utilized to classify pictures of 0s or 2s. The “extra” portion of the dataset was used bringing the total number of training points to 129,090 and 5,893 testing points, each with 648 features [23]. The second dataset is the much larger, and much more difficult HIGGS dataset [2]. The HIGGS dataset contains several features measured by particle detectors during experiments in the Large Hadron Collider, along with a classification as to if a Higgs Boson was indicated in that experiment. The training set consists of 10,500,000 data points while the testing set consists of 500,000 data points each with 28 features.

For the first experiment with the SVHN dataset, we trained a neural network with two hidden layers and ReLU activation functions. The first hidden layer had 100 nodes while the second had 50. This problem was fairly easy for a neural net to solve as the testing accuracy quickly rose near to its final value. However, the problem did not require much data and gradient methods were suitable to solve the problem sufficiently. This made the problem well suited to GPU based implementations. Although the GPU methods typically reached convergence faster (Figure 10b), the ADMM method provided promising results. The experiment demonstrated ADMM scaled linearly with an increase in the number of nodes (Figure 10a), allowing it to compete with GPUs when a sufficient number of cores had been used. Every gradient-based approach has shown diminishing returns with additional cores, until the time needed to optimize actually begins to rise. In comparison, ADMM neatly takes approximately half the time when distributed across twice the cores.

In figure 10b, we demonstrate the competitive performance of ADMM by graphing each methods test set accuracy as a function of time. With 1024 cores, ADMM was able to meet the 95% threshold in 13.3 seconds. After an extensive hyperparameter search, the GPU methods did demonstrate faster convergence with SGD converging at an average of 28.3 seconds, L-BFGS in 3.3 seconds, and conjugate gradients in 10.1 seconds. With the small dataset, we were unable to use enough cores to overcome the benefits of the GPU, but were nonetheless able to demonstrate its competitiveness.

The second experiment was performed on the much larger and more difficult Higgs dataset. For this task, we optimized a neural net with a hidden layer of 300 nodes with ReLU activation functions, as suggested by [2]. The graphs demonstrate the time to an accuracy threshold of 64%. This value was chosen as it was a threshold that all of the methods could reliably meet when tested over many trials. Figure 11a demonstrates the scalable nature of ADMM by drastically reducing the amount of training time required with an increase in cores and again demonstrating strong linear scaling properties.

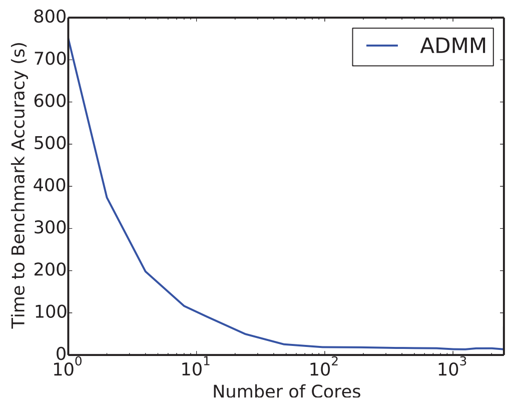
For this much more difficult problem, the scaling properties of ADMM allow it to reach convergence much faster than the alternative methods. ADMM reaches convergence the fastest in 7.8 seconds as shown in figure 11b. In contrast, L-BFGS reached convergence in 181 seconds[2] while conjugate gradients required 44 minutes. SGD never reached the 64% threshold on the testing set in 7 hours of training. For very large, difficult problems, these results suggest that the linear scaling of ADMM enables this method to leverage the large number of cores to dramatically out-perform GPU implementations.

## 4 Future Work

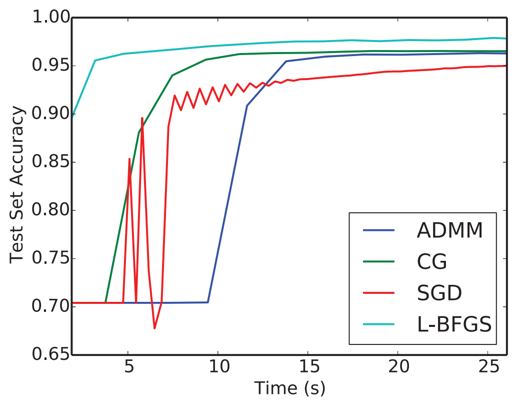
We believe the ADMM method implemented was a good first attempt in trying to make CPUs a more viable source for implementers of neural and deep network architectures. However, there are limitations in our current algorithm which should be explored. In this section, we discuss some of those limitations along with ideas for directions of research.

First, the number of features in a layer had to be limited due to the inverse operation on the otherwise large $\phi_l \phi_l^T$ matrix which appears in the update of the matrix $W_{l+1}$. Each node $i$ must compute and communicate its component $\phi_l^i \phi_l^{iT}$; the sum of these must then be inverted. Normally, $k_{l+1}$ is much smaller than the number of data points, making this communication and $O(k_{l+1}^3)$ inversion fast. However, this assumption is not always valid, making this step expensive or memory prohibitive. Second, the algorithm does not seem to suit large deep network architectures. The algorithm is deterministic and will ensure the

[2]It is worth noting that though L-BFGS required substantially more time to reach 64% than ADMM, it was the only method to produce a superior classifier, doing as well as 75% accuracy on the test set



(a) **Time required for ADMM to reach 95% test accuracy when parallelized over varying numbers of cores**. This problem was not large enough to support parallelization over many cores, yet the advantages of scaling are still apparent (note the x-axis is scaled logarithmically). However, in comparison, on the GPU, L-BFGS reached this threshold in 3.2 seconds, conjugate gradients in 9.3 seconds, and stochastic gradient descent in 8.2 seconds



(b) **Test set predictive accuracy as a function of time in seconds** for ADMM on 2,496 cores (blue), in addition to GPU implementations of conjugate gradients (green), SGD (red), and L-BFGS (cyan).

Figure 10: ADMM Street View House Numbers Results

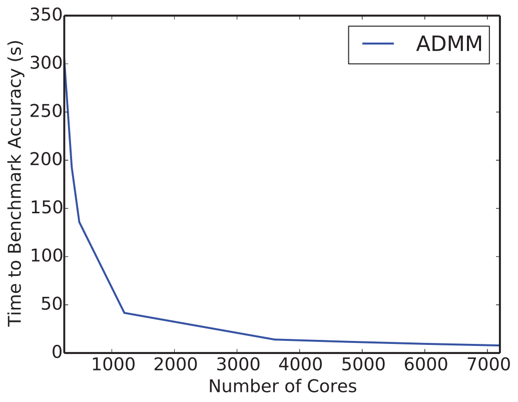
error always decreases in value. As a result, in the complex spaces induced by deep networks, ADMM frequently very quickly settles on a local minimum which is far from globally optimal. We discuss the introduction of stochasticity in an attempt to avoid these local minima.

## 4.1 Increasing the Number of Features

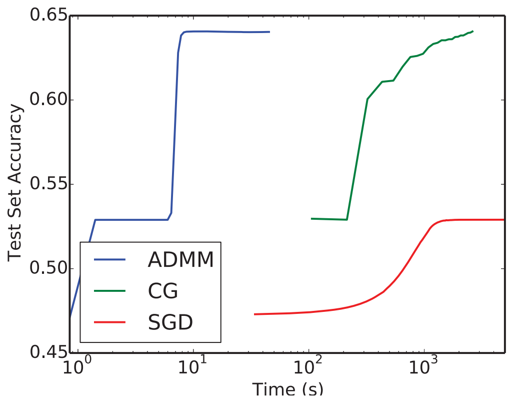
The current ADMM method works in a distributed manner allowing for much of the work to be done on many compute nodes, as is illustrated in Algorithm 1. However, if the number of features in a given layer $k_{l+1}$ is large, the current algorithm breaks down due to the $O(k_{l+1}^2)$ storage requirement in every node due to the computation of $\phi_l \phi_l^T$ required in the weight update, and the $O(k_{l+1}^3)$ matrix inversion computation of a $k_{l+1} \times k_{l+1}$ matrix. This makes running this algorithm with a large number of features computationally expensive and often memory prohibitive. However, the algorithm can be improved by utilizing the Nyström and Woodbury approximations [17] as long as the $\phi_l \phi_l^T$ matrix is low-rank.

### 4.1.1 Low-rank Requirement

A matrix is said to be rank $r$ if there a set of $r$ columns for which the span of the selected columns and the original matrix are the same; if $r$ is much smaller than the actual number of columns $k$, the matrix is said to be "low-rank." Low-rank matrices can often be approximately



(a) **Time required for ADMM to reach 64% test accuracy when parallelized over varying numbers of cores**. L-BFGS on a GPU required 181 seconds, and conjugate gradients required 44 minutes. SGD never reached 64% accuracy.



(b) **Test set predictive accuracy as a function of time** for ADMM on 7200 cores (blue), conjugate gradients (green), and SGD (red). Note the x-axis is scaled logarithmically.

Figure 11: ADMM Higgs Results

reconstructed from a smaller subset of their columns. In scenarios where matrices are full rank, the matrix cannot be well approximated with a subset of columns as all columns of the matrix are required to produce the span defined by the original matrix. Therefore, the Nyström and Woodbury approximations only become useful if $\left(\phi_l \phi_l^T\right)^{-1}$ is of rank $r$, where $r \ll k$.

### 4.1.2 Algorithm

For simplicity of discussion, we describe the algorithm for a neural net consisting of a single neuron - it expands easily to more complex networks.

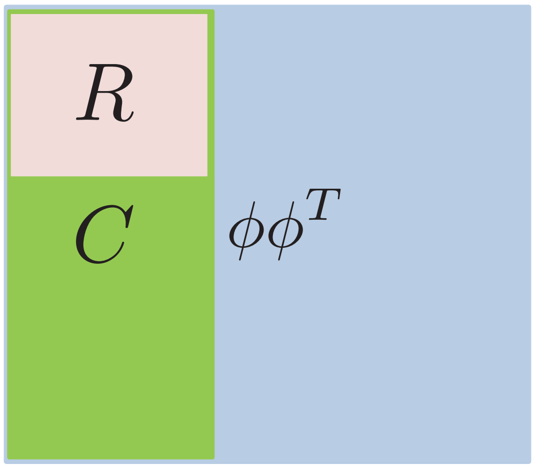


Figure 12: A depiction of the matrices $C$, $R$, and $\phi\phi^T$ where $R \subseteq C \subseteq \phi\phi^T$. $R$ is $p \times p$, $C$ is $k \times p$, and $\phi\phi^T$ is $k \times k$.

**Algorithm 2** New ADMM Method

```
 1: for each node $i \in N$, given $\phi^i$, $t^i$, and $W$ do
 2:     Compute $C_i$
 3: end for
 4: if server node then
 5:     $C = \sum_i C_i$
 6:     $R \leftarrow C[:l][:l]$
 7: end if
 8: while not converged do
 9:     for each node $i \in N$ do
10:         Compute $z^i\phi^{iT}$
11:     end for
12:     if server node then
13:         $z\phi^T = \sum_i z^i\phi^{iT}$
14:         $W = z\phi^T C(R^{-1} + C^T C)^{-1} C^T$
15:     end if
16: end while
```

The goal of the new algorithm, Algorithm (2), will be to avoid this expensive $O(k^2)$ storage and $O(k^3)$ computation. In order to do so, two new matrices are introduced, $R$ and $C$. Let $R \subseteq C \subseteq \phi\phi^T$, where $C$ is defined as a $k \times p$ matrix ($p \ll k$), and $R$ is defined to be $p \times p$ (see Figure 12). Ideally, $\phi\phi^T$ is low rank, and the span of the columns in $C$ would be equivalent to the span of the columns in $\phi\phi^T$. A sampling technique is utilized to choose appropriate columns to comprise $C$.

The Nyström approximation states that the $\phi\phi^T$ matrix can be approximated by computing $CR^{-1}C^T$. The Woodbury method gives the equality $(CR^{-1}C^T)^{-1} = C(R^{-1}+C^TC)^{-1}C^T$.

By utilizing both methods, the avoidance of computing and inverting a $k \times k$ matrix is made. The largest matrix that now has to be computed is a $k \times p$ matrix where $p \ll k$ (2).

This approach promises to save large amount of communication and computation time for problems with large feature sets.

### 4.2 Application to Deep Nets

While ADMM performs well on shallow networks, additional work will need to be done in order to allow for the application of ADMM to deep nets. The ADMM method allows for quick convergence because it is guaranteed to decrease error in non-convex spaces, but this also negatively impacts the algorithm by making it extremely likely to end up in a local minimum. The introduction of stochasticity may alleviate the problem by allowing for random steps that do not necessarily decrease the error function but do, potentially, get us closer to the global minimum as opposed to one of the many local minima. Stochastic ADMM has shown success in non-convex contexts other than neural nets [25].

There are several approaches we can take to solve this problem including training on random subsets of data. Exploration into these topics will need to be done to see where stochasticity can be introduced and where it is effective. In any case, it should help improve the algorithm by allowing for a solution to the local minima problem.

## 5 Conclusion

Neural nets have shown remarkable success in classification and regression problems. However, there is much improvement that needs to be made in order to speed up the rate at which training can occur. Through the introduction of kernel based weight initialization and an improved ADMM method, we are showing how to speed up the training process while maintaining at least the same level of prediction accuracy. Informed weight initialization provides the opportunity to supplement a layer in deep nets with a nearly-trained set of weights, while ADMM allows for much faster training in a highly distributed fashion. Furthermore, while we have shown improvement, there are still improvements than can be made by allowing for greater numbers of features in training and applying ADMM to deep nets.

## References

[1] A Aizerman, Emmanuel M Braverman, and Li Rozoner. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25:821–837, 1964.

[2] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature Communications*, 5, 2014.

[3] Christopher M Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., 2006.

[4] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, pages 144–152. ACM, 1992.

[5] Léon Bottou. Stochastic gradient learning in neural networks. In *Proceedings of Neuro-Nimes*, 1991.

[6] Olivier Bousquet and Léon Bottou. The tradeoffs of large scale learning. In *Advances in Neural Information Processing Systems*, pages 161–168, 2008.

[7] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Andrew Y Bradski, Gary adn Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems*, 2007.

[8] George Cybenko. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[9] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition*, volume 1, pages 886–893. IEEE, 2005.

[10] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Pual Tucker, Ke Yang, Quock V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[11] Ken-Ichi Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural networks*, 2(3):183–192, 1989.

[12] Xavier Glorot and Yoshua Bengio. Understanding the difficult of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[13] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1026–1034, 2015.

[15] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[17] Sanjiv Kumar, Mahryar Mohri, and Ameet Talwalkar. Sampling methods for the nyström method. *The Journal of Machine Learning Research*, 13(1):981–1006, 2012.

[18] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[19] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural Networks: Tricks of the Trade*, pages 9–48. Springer, 2012.

[20] Ryan Mcdonald, Mehryar Mohri, Nathan Silberman, Dan Walker, and Gideon S Mann. Efficient large-scale distributed training of conditional maximum entropy models. In *Advances in Neural Information Processing Systems*, pages 1231–1239, 2009.

[21] James Mercer. Functions of positive and negative type, and their connection with the theory of integral equations. In *Philosophical Transactions of the Royal Society of London. Series A, containing papers of a Mathematical or Physical Character*, volume 209, pages 415–446. JSTOR, 1909.

[22] Dmytro Mishkin and Jiri Matas. All you need is a good init. *International Conference on Learning Representations*, 2016.

[23] Yuval Netzer, Tao Want, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, volume 2011, page 5. Granada, Spain, 2011.

[24] Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, Quoc V Le, and Andrew Y Ng. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 256–272, 2011.

[25] Hua Ouyang, Niao He, Long Tran, and Alexander Gray. Stochastic alternating direction method of multipliers. In *Proceedings of the 30th International Conference on Machine Learning*, pages 80–88, 2013.

[26] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *Advances in Neural Information Processing Systems*, pages 1177–1184, 2007.

[27] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 873–880. ACM, 2009.

[28] Carl Edward Rasmussen. Gaussian processes for machine learning. 2006.

[29] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386, 1958.

[30] Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, and Andrew Cotter. Pegasos: Primal estimated sub-gradient solver for svm. *Mathematical Programming*, 127(1):3–30, 2011.

[31] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dorpout: A simple way to prevent neural networks from overfitting. In *The Journal of Machine Learning Research*, volume 14, pages 1929–1958. JMLR, 2014.

[32] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[33] Li Wan, Matthew Zeiler, Sixin Zhang, Yann L Cun, and Rob Fergus. Regularization of neural netowrks using dropconnect. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1058–1066, 2013.

[34] Martin Zinkevich, John Langford, and Alex J Smola. Slow learners are fast. In *Advances in Neural Information Processing Systems*, pages 2332–2339, 2009.

[35] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 2595–2603, 2010.